X-KAAPI: a Multi Paradigm Runtime for Multicore Architectures

Thierry Gautier*, Fabien Lementec*, Vincent Faucher†, Bruno Raffin*
*INRIA
Email: thierry.gautier@inrialpes.fr, fabien.lementec@inria.fr,bruno.raffin@inria.fr
† CEA, DEN, DANS, DM2S, SEMT, DYN, Gif-sur-Yvette, F-91191
Email: vincent.faucher@cea.fr

***Abstract*—The paper presents X-KAAPI, a compact runtime for multicore architectures that brings multi parallel paradigms (parallel independent loops, fork-join tasks and dataflow tasks) in a unified framework without performance penalty. Comparisons on independent loops with OpenMP and on dense linear algebra with QUARK/PLASMA confirm our design decisions. Applied to EUROPLEXUS, an industrial simulation code for fast transient dynamics, we show that X-KAAPI achieves high speedups on multicore architectures by efficiently parallelizing both independent loops and dataflow tasks.**

I. INTRODUCTION

Industrial codes usually require mixing different parallelization paradigms to achieve interesting speedups. The challenge is to develop programming and runtime environments that efficiently support this multiplicity of paradigms. We introduce X-KAAPI, a runtime for multicore architectures designed to support multiple parallelization paradigms with high performance thanks to a low overhead scheduler. The proposed case study is the industrial numerical simulation code for fast transient dynamics called EUROPLEXUS (abbreviated EPX in the following paragraphs). EPX [1,2] is dedicated to complex simulations in industrial framework, with a large source code composed of 600.000 lines of Fortran. It supports 1-D, 2-D and 3-D models, based on either continuous or discrete approaches, to simulate structures and fluids in interaction. EPX supports non-linear physics for both geometrical (finite displacements, rotations and strains) and material (plasticity, damage, etc) properties. A typical simulation spends more than 70% of the execution time in:

1) large loops with independent iterations,
2) Sparse Cholesky matrix factorizations in a Skyline storage format.

As EPX is mainly used to simulate impacts and explosions, the considered systems undergo important modifications during the simulation (large structural displacements and strains for instance), leading to changing and unbalanced work loads.

Reaching high performance on multicore architectures requires several threads of control running mostly independent code, with few synchronizations to ensure a correct progress of the computation. Programming directly with threads is highly unproductive and error prone [20]. Two main programming alternatives have been developed. Cilk [4] promotes a fork-join parallel paradigm with theoretical guarantees on the expected performance. OpenMP [23] relies on code annotations to generate parallel programs. Cilk and OpenMP have both basic constructs to parallelize independent loops. With OpenMP the user has also the ability to guide the way iterations are scheduled among the threads.

Ten years after Cilk, the introduction of tasks in OpenMP-3.0 makes Cilk and OpenMP, at a first glance, very close. They seem to be good candidates for a task based Cholesky factorization [19], [1], but the current implementation of tasks in OpenMP-3.0 [23] (in Intel compiler or GCC compiler) is several orders of magnitude more costly than in Cilk, making it hard to reach portable performance.

Moreover, [19], [10] show that OpenMP-3.0 and Cilk parallel models limit the available parallelism for a dense Cholesky factorization. The authors promote a data flow runtime that is able to encode finer data flow synchronizations between tasks. The runtime can detect concurrent tasks as soon as their inputs are produced. Such data flow programming model is a promising approach for our sparse Cholesky factorization.

Several runtimes and languages were based on a data flow paradigm, like Athapascan [14] used for sparse Cholesky factorizations [9], QUARK [28], the data flow runtime of the PLASMA dense linear algebra library [8], the StarSS programming model with its SMP implementation called SMPSs [3] or OMPSs [7], or StarPU [1] dedicated to multi-GPU computations. But none of these softwares support independent loops. Moreover, except recently in OMPSs, they do not allow the creation of recursive tasks, discarding recursive parallel algorithms.

The X-KAAPI runtime we introduce in this paper proposes a new unified framework based on data flow tasks and work-stealing dynamic scheduling to develop multi-paradigms fine grain parallel programs. A comparison with OpenMP shows that our dynamic scheduler can outperform both the static and dynamic OpenMP scheduler. We also used X-KAAPI to develop a binary compatible QUARK library to schedule PLASMA's algorithms with better scalability at a finer grain. Finally, we report preliminary results mixing both parallel loops and data flow task parallelism in EPX.

Next section presents the X-KAAPI's parallel programming model. We focus on its adaptive task model, and how it is used to support parallel loops. Section III reports experimental evaluations compared to OpenMP [23] on parallel loops and QUARK [28] on dense Cholesky factorizations.

[1] http://europlexus.jrc.ec.europa.eu/public/manual_html/index.html
[2] http://www.repdyn.fr

Section IV evaluates the parallelization of EPX as compared with OpenMP, before concluding.

II. Data flow task programming with X-Kaapi

The X-Kaapi's task model [15] , as for Cilk [4], Intel TBB [25], OpenMP-3.0 [23] or StarSs/SMPSs/OMPSs [3], [7], enables non blocking task creation: the caller creates the task and continues the program execution. The semantic remains sequential like for its predecessor Athapascan [14], but the runtime was redesigned [15] and the task model extended to support adaptive tasks (section II-D).

The execution of a X-Kaapi program is a multi-threaded process. It starts by the creation of a poll of threads responsible to execute the tasks generated at runtime. By default, the X-Kaapi runtime creates on thread per core of a multi-core machine. The execution of a X-Kaapi program generates a sequence of tasks that access to data in a shared memory. From this sequence, the runtime extracts independent tasks to dispatch them to idle cores. We focus here on the multicore version of X-Kaapi.

A. Design choices

More than a runtime, X-Kaapi[3] is a fully featured software stack to program heterogeneous parallel architectures. The stack is written in C and was designed using a bottom up approach: each layer is kept as specialized as possible to fit a specific need. Currently, the stack includes: a runtime supporting multicores and multiprocessors; a set of ABIs (QUARK [28], OpenMP runtime libGOMP); a set of high level APIs (C [21], Fortran and C++; subset of Intel TBB [25]); and a source to source compiler [22] based on the ROSE framework [24].

B. Data flow task model

A X-Kaapi program is composed of sequential C or C++ code and some annotations or runtime calls to create tasks. The parallelism in X-Kaapi is explicit, while the detection of synchronizations is implicit [15]: the dependencies between tasks and the memory transfers are automatically managed by the runtime.

A task is a function call that returns no value except through the shared memory and the list of its effective parameters. Depending of the APIs, tasks are created using code annotation (`#pragma kaapi task` directive) if the X-Kaapi's compiler [22] is used, or by library function (`kaapic_spawn` call using X-Kaapi's C API [21]), or by low level runtime function calls.

Tasks share data if they have access to the same memory region. A memory region is defined as a set of addresses in the process virtual address space. This set has the shape of a multi-dimensional array. The user is responsible for indicating the mode each task uses to access memory: the main access modes are *read*, *write*, *reduction* or *exclusive* [14], [15], [22], [21]. When required [15], the runtime computes true dependencies (Read after Write dependencies) between tasks thanks to the access modes. At the expense of memory copy, the scheduler may solve false dependencies through variable renaming.

[3]http://kaapi.gforge.inria.fr

A thread that performs a task may creates child tasks and pushes them in its own workqueue. The workqueue is represented as a stack. The enqueue operation is very fast, typically about ten cycles on the last x86/64 processors. As for Cilk, a running X-Kaapi's task can create child tasks, which is not the case for the other data flow programming softwares previously mentioned [28], [3], [1]. Once a task ends, the runtime executes the children following a FIFO order. During task execution, if a thread encounters a stolen task, it suspends its execution and switches to the workstealing scheduler that waits for dependencies to be met before resuming the task. Otherwise, and because sequential execution is a valid order of execution [14], [15], tasks are performed in FIFO order without computation of data flow dependencies.

C. Execution with workstealing algorithm

X-Kaapi relies on *workstealing*, popularized by Cilk [4], to dynamically balance the work load among cores. Once a thread becomes idle, it becomes a thief and initiates a steal request to a randomly selected victim. On reply, the thief receives one or more ready tasks. X-Kaapi favors *request aggregation* [17]: N pending requests from N thieves to a same victim are handled in one operation, reducing the number of ready task detections. A theoretical analysis in [26] shows a reduction of the total steal request number. In our protocol, one of the thieves is elected to reply to all requests.

As opposed to Cilk, X-Kaapi considers tasks with data flow dependencies. Following the *work first principle* [13], X-Kaapi computes ready tasks only during steal operation, favoring work at the expense of the critical path. The detection of a ready task consists in a traversal of the victim stack from the top most task (the oldest), to look all its predecessors have been completed. X-Kaapi synchronizes the thief and victim following the Cilk's T.H.E protocol [13]. Except in rare cases, the victim and the thief execute concurrently. Using this approach, X-Kaapi and Cilk show similar overheads for the execution of independent tasks (see section III-A).

The overhead to manage tasks and to compute the data flow graph may remain important. To reduce this overhead, X-Kaapi implements two original optimizations.

First, when the cost of computing ready tasks becomes important, the runtime attaches to the victim an accelerating data structure for steal operations. The structure contains a list that gets updated with tasks becoming ready due to the completion of their data flow dependencies. A subsequent steal operation is reduced to the pop of a task from the ready list (nearly constant time operation), without a traversal of the victim stack.

The second optimization enables a more fundamental reduction of parallelism overhead. Parallel versions of some algorithms require more operations than their sequential counterpart. The overhead is directly related to the number of created tasks. The idea is thus to limit the number of tasks by creating them on demand, as computing resources become idle. These so called *adaptive* tasks are detailed in the following section.

D. Adaptive task model

Writing performance-portable programs within the task programming model requires creating much more tasks than available computing resources. Then, the scheduler can efficiently and dynamically balance the work load. However, the extra operations required to merge the partial results account for overhead since it is not present in the sequential algorithm. Fich [12] proved that any parallel algorithm of time $\log n$ to compute prefix of n inputs requires at least $4n$ operations, versus $n-1$ operations in sequential. Adapting the number of created parallel tasks to dynamically fit the number of available resources is the key point to reach high performance. With an other approach for implementing this adaptation, we have proposed this on demand task creation to build coarse grain parallel adaptive algorithms for most of the STL algorithms [27]. Here, the proposed solution extends the task model for a much more finer integration with the scheduler.

In data flow model, once all inputs of a task are produced, it becomes ready for execution. A task being executed cannot be stolen. To allow on demand task creation, X-KAAPI extends this model: a task publishes a function, called the *splitter*, to further divide the remaining work. The splitter is called on a running task by an idle thread during a steal operation. The task and its splitter are concurrent and must be carefully managed as they both need to access shared data structure. The programmer is held responsible for writing correct task and splitter codes. To help him, the X-KAAPI runtime ensures that only one thief performs splitter concurrently with the task execution. It allows for simple and efficient synchronization protocols. Moreover, for applications developers, a set of higher parallel algorithms, like those of the STL [27], are proposed on top of the adaptive task model. Next section focuses on the parallel *foreach* algorithm.

E. Adaptive tasks for parallel loops

Following the OpenMP `parallel for` directive, X-KAAPI proposes a parallel loop function called `kaapic_foreach`, which is used in the backend of our X-KAAPI compiler [22].

A call to `kaapic_foreach` creates an adaptive task that iterates through the input interval $[first, last)$ to apply a functor (the loop body). The initial interval is partitioned in p slices, one slice reserved to each available core. When a thread calls the splitter to obtain work from the adaptive task, it grabs the reserved slice if available. The splitter returns an adaptive task that calls the functor for each iteration of the slice.

If the initial slice is not longer available, the splitter tries to split the interval $[b_t, e)$ corresponding to the iteration that remains to be process at time t. Thanks to the concurrency level guaranteed by the scheduler, a T.H.E like protocol [13] ensures coherent split of interval while task iterates. The aggregation protocol is able to process k steal requests at once. The main thief tries to split $[b_t, e)$ into $k+1$ equal slices, leaving one slice for the victim. Then, for each of the k requests, the thief returns approximately the same amount of work for balancing purpose [26].

III. BENCHMARKS

This section presents a synthetic selection of three benchmarks to compare X-KAAPI performance with respect to three parallel programming models: fork-join model, parallel loops and data flow tasks.

The multicore platform used in this section is a 48 cores AMD ManyCours platform with 256GBytes of main memory. Each core frequency is 2.2Ghz. The machine has 8 NUMA nodes. Each node has 6 cores sharing a L3 cache of 5 MBytes. Reported times are averaged over 30 runs.

A. Task creation time

This section compares the overhead of task creation and execution with respect to the sequential computation. The experiment evaluates the time to execute the X-KAAPI program of figure 1 for computing the 35-th Fibonacci number. The program recursively creates tasks without any data flow dependency. X-KAAPI is compared with Intel Cilk+ (icc 12.1.2), Intel TBB-4.0 and OpenMP-3.0 (gcc-4.6.2). Fibonacci is a standard benchmark used by Cilk [13] and Intel TBB (part of TBB-4.0 source code). Sequential time is $0.091s$. Figure 1 reports times using 1, 8, 16, 32 and 48 cores.

```
void fibonacci(long* result,
               const long n)
{
  if (n<2)
    *result = n;
  else
  {
    long r1,r2;
#pragma kaapi task write(&r1)
    fibonacci( &r1, n-1 );
    fibonacci( &r2, n-2 );
#pragma kaapi sync
    *result = r1 + r2;
  }
}
```

#cores	Cilk+	TBB	Kaapi	OpenMP
1 (slowdown:1)	1.063 (x 11.7)	2.356 (x 26)	0.728 (x 8)	2.429 (x27)
8	0.127	0.293	0.094	*51.06*
16	0.065	0.146	0.047	*104.14*
32	0.035	0.072	0.024	*(no time)*
48	0.028	0.049	0.017	*(no time)*

Fig. 1. Top. Fibonacci micro benchmark. Bottom. Execution times (in seconds). Sequential time is 0.091s for Fibonacci 35.

Benchmark sources for OpenMP or TBB are not listed but they create exactly the same number of tasks and synchronization points. TBB has more overhead with respect to the sequential computation (slowdown of about 26) in comparison to X-KAAPI (slowdown of 8). This overhead can easily be amortized by increasing the task granularity, but at the expense of increasing the critical path, thus reducing the available parallelism [13]. OpenMP (gcc 4.6.2) performs poorly: the grain is too fine and OpenMP cannot speed up the computation. Computation was stopped on 32 and 48 cores after 5 minutes. The relatively good time of OpenMP with 1 core is due to an artifact of the libGOMP runtime: for one core, task creation is degenerated to a standard function call.

B. Data flow Cholesky factorization

The Cholesky factorization is an important algorithm in dense linear algebra. This section reports performances of the block version `PLASMA_dpotrf_Tile` of PLASMA 2.4.6 [8]. On a multicore architecture, PLASMA relies on the runtime QUARK [28] to manage data flow tasks. QUARK only supports a subset of the functionalities offered by X-KAAPI. Thus, we have ported QUARK on top of X-KAAPI to produce a binary compatible QUARK library, which is linked with PLASMA algorithms for X-KAAPI experiments.

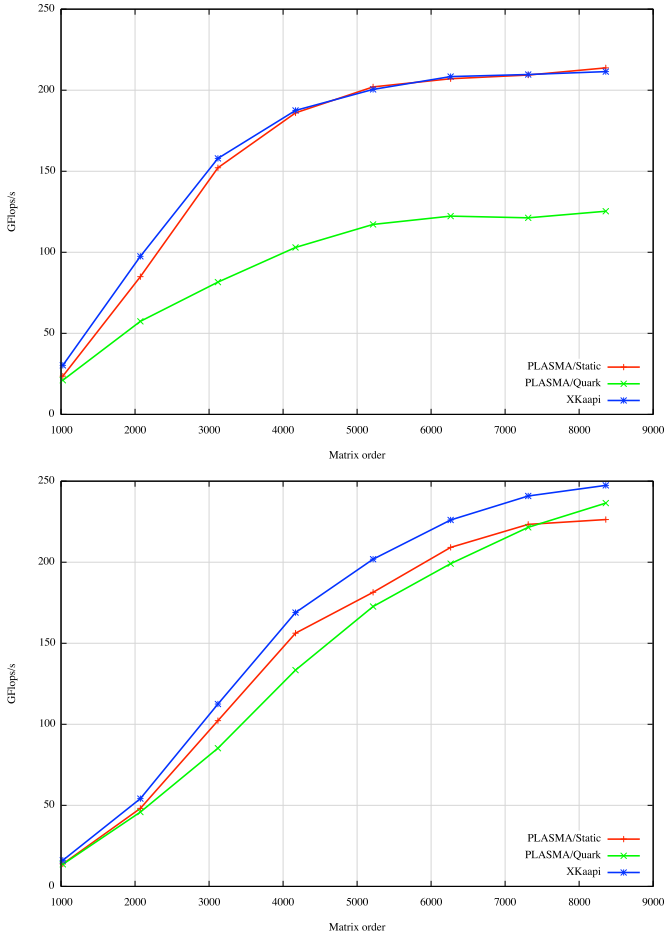


Fig. 2. Gflops on Cholesky algorithm with QUARK and X-KAAPI. Tile size of $NB = 128$ (top) or $NB = 224$ (bottom).

Figure 2 reports the performances (GFlop/s) with respect to the matrix size on 48 cores of the three available versions : the first two versions are based on tasks: QUARK (labelled PLASMA/Quark) and X-KAAPI (labelled XKaapi) versions; the last version is a statically parallelized algorithms in PLASMA (labelled PLASMA/static) that does not incur overhead in task management.

QUARK implements a centralized list of ready tasks, with some heuristics to avoid accesses to the global list. For fine grain tasks ($NB = 128$) and due to a contention point to access the global list, X-KAAPI outperforms QUARK. We can expect this contention point to become more severe as the core number increases with next generation machines, affecting PLASMA performance. When the grain increases, X-KAAPI remains better but the difference decreases because of the relatively small impact of task management with respect to the whole computation. One can also note that increasing the grain size reduces the average parallelism and limits the speedup. For matrix of size 3000, the performance for $NB = 128$ reaches $150 GFlops$, while for $NB = 224$, it drops to about $105 GFlops$.

For fine grain tasks ($NB = 128$), our X-KAAPI implementation has less overhead than QUARK implementation. X-KAAPI obtains performances closed to the statically parallelized PLASMA algorithm. For larger grain, both tasks' based implementation (X-KAAPI or QUARK) better balance the workload among the core. Due to the NUMA nature of the architecture, some tasks are longer to execute on some ressource because of remote data accesses. The execution is not regular and our X-KAAPI implementation allows to balance the workload between cores.

C. Parallel independent loops

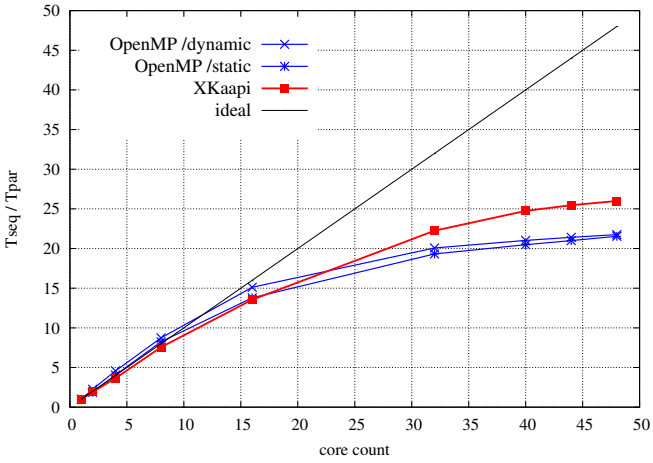


Fig. 3. Comparison of parallel loop speedup

We compare OpenMP/GCC 4.6.2 parallel loop using static and dynamic schedule against X-KAAPI (`kaapic_foreach` version). Figure 3 reports speedups of the two parallel loops of the EPX application (next section). Both OpenMP static and dynamic schedule have the same performances. Globally, OpenMP and X-KAAPI speedups are very close, but X-KAAPI outperforms OpenMP past 25 cores. The same cores were used by X-KAAPI and OpenMP by binding threads to cores using an affinity mask.

IV. IMPLEMENTATION AND TESTS WITH EPX

EPX is a computer program for the simulation of fluid-structure systems under transient dynamic loading. The code is co-owned since 1999 by French CEA and European Commission (Joint Research Center, Institute for the Protection ans Security of the Citizen) and jointly developed through a consortium also involving EDF (French national electricity board) and ONERA (French aerospace research labs). EPX uses finite elements, SPH particles or discrete elements to model structures and finite elements, finite volumes or SPH

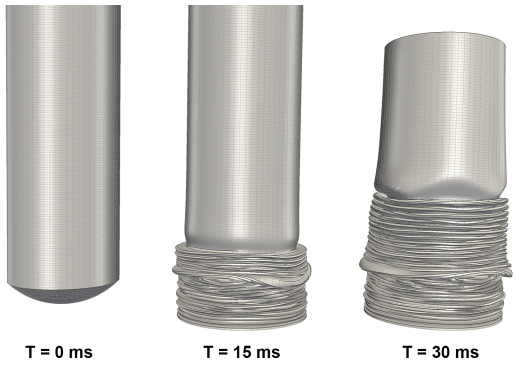


Fig. 4. The MEPPEN simulation.

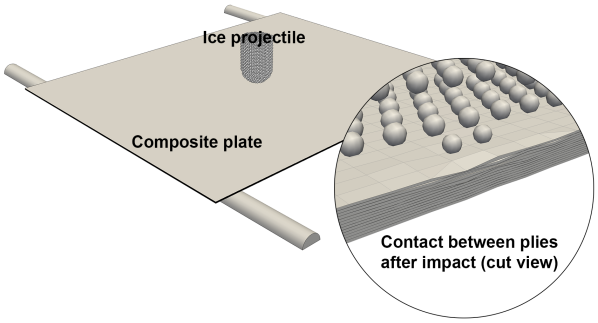


Fig. 5. The MAXPLANE simulation.

particles to model fluids. EPX is dedicated to simulating the consequences of extreme loadings such as explosion and impacts, with strong coupling between structures and fluids. Time integration is explicit (central difference schemes for structures and explicit Euler for fluids) and about 140 geometric elements are available, along with about 100 material models and about 50 kinds of kinematic connexions between entities, such as unilateral contact, fluid structure links for both conformant meshes and immersed boundaries or various kinds of constrained motions. To avoid non-physical parameters throughout the kinematic constraints enforcement procedure, Lagrange multipliers are used to compute link forces, yielding the need for linear system solvers alongside the classical explicit solution process.

The source code is thus complex (600.000 lines of Fortran) and many algorithms are involved simultaneously within classical simulations. However, two main kinds of algorithmic tasks accounts for 70% of a common EPX execution: 1/ independent parallel loops for nodal force vector evaluations and kinematic link detection; 2/ sparse Cholesky factorization of the so-called **H** matrix, obtained from the condensation of dynamic equilibrium equations onto Lagrange multipliers, in a Skyline representation (the cost of following triangular system solutions being neglected).

Practically, three main algorithms are considered for subsequent examples: representative of classical EPX simulations in structural domain. They are named from the Fortran procedure used in EPX to perform the task:

1) LOOPELM: independant loop on finite elements to compute nodal internal forces from local mechanical behaviour,
2) REPERA: independant loop to sort candidates for *node_to_facet* unilateral contact,
3) CHOLESKY: perform Cholesky factorization of a symetric positive semi-definite matrix.

The distribution of execution times between these three algorithms varies with the simulation step and with the considered instance. In this paper we focus on two simulation scenarios. The first one, called MEPPEN (see figure 4), consists in the crash of a large steel missile on a perfectly rigid wall. The second one, called MAXPLANE (see figure 5), consists in the impact of a ice projectile on a composite plate. Due to physics and modelling, these two instances provide very different repartitions of time among the considered algorithms:

- MEPPEN is characterized by large structural strains, strongly non-linear behaviour and multiple contacts as the missile undergoes dynamic buckling: time is then mainly split between LOOPELM, with large ratios between finite elements, and REPERA,
- MAXPLANE is characterized by a modelling of the composite plate plies using 3D finite elements, with contact conditions between each plies, so that the size and filling of the **H** matrix are close to those of the system stiffness matrix: the solution procedure is then strongly dominated by the condensed system solution, and then by the CHOLESKY algorithm.

A. LOOPELM and REPERA loops

Figure 6 details the X-KAAPI speedups for the MEPPEN (left) and MAXPLANE (right) instances. On the smallest instance, MEPPEN, the LOOPELM has limited speedup due to its memory intensive character. REPERA is more computation intensive leading to a good speedup.

B. Sparse Cholesky factorization

The sparse Cholesky factorization (LDL^t) represents about 60% of the execution time for the MAXPLANE instance. The numerical scheme requires to factor and solve a linear system at each time step. The linear system is sparse and its size and density depend on the interactions in the simulation. The pseudo sequential code is sketch in figure 7. Variable `sli` is the skyline representation of the sparse matrix to factorize. The function calls `potrf`, `trsm`, `syrk` and `dgemm` at lines 3, 7, 12, and 17 are pseudo blas functions with `sli` the skyline matrix parameter and k, n, m the indexes delimiting the block to process. All these calls create tasks in the X-KAAPI version. Only calls at line 7, 12 and 17 create tasks in OpenMP.

In the X-KAAPI version, these indexes serves as defining memory accesses to compute dependencies. OpenMP parallelization implies synchronization between tasks in order to satisfy data flow dependencies. So, `#pragma omp taskwait` directives have to put after lines 8 and 19. As noted by [19], the parallel data flow version only specifies tasks with access modes, without explicit synchronizations.

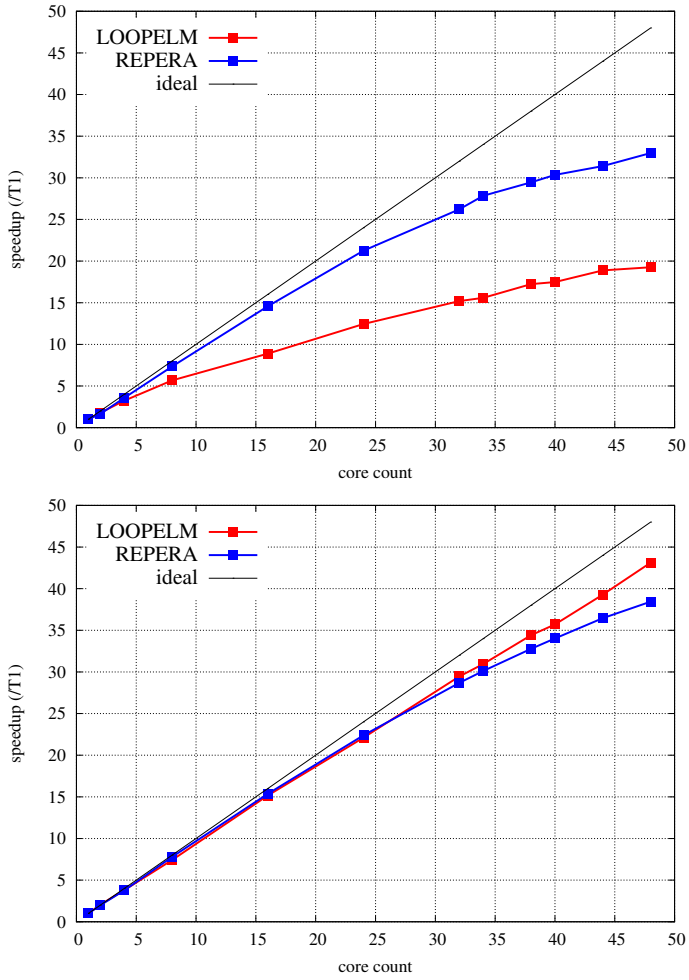


Fig. 6. Speedups of LOOPELM and REPERA on MEPPEN and MAXPLANE.

Figure 7 reports speedup using a matrix that appear during the MAXPLANE simulation. The dimension of the matrix is 59462 with 3.59% of non zero elements. We looked for the best block size for this experiment: $BS = 88$. The sequential time is $47.79s$. X-KAAPI version outperforms OpenMP (gcc-4.6.2) version, for the same reasons as for the dense Cholesky factorization [19].

C. Overall gains for EPX

Figure 8 reports the performances of the parallel version with respect to the sequential code. The left bar in the histograms represents sequential time decomposition with respect to the time of each algorithm presented above. The Amdhal's law applies: 'Other' part (30%) is being analyzed for parallelization.

V. RELATED WORK

Kaapi [15] was designed in our group after the preliminary work on Athapascan [14], [9]. X-KAAPI keeps definition of access mode to compute data flow dependencies between a sequence of tasks. StarSs/SMPSs [3], QUARK [28], StarPU [1] follow the same design. Differences are in the

```
1  for (k = 0; k < N; k += BS)
2  {
3    potrf(k, &sli);
4    for (m = k + BS; m < N; m += BS)
5    {
6      if (is_empty(m, k, &sli)) continue;
7      trsm(k, m, &sli);
8    }
9    for (m = k + BS; m < N; m += BS)
10   {
11     if (is_empty(m, k, &sli)) continue;
12     syrk(k, m, &sli);
13     for (n = k + BS; n < m; n += BS)
14     {
15       if (is_empty(n, k, &sli)) continue;
16       if (is_empty(m, n, &sli)) continue;
17       gemm(k, m, n, &sli);
18     }
19   }
20 }
```

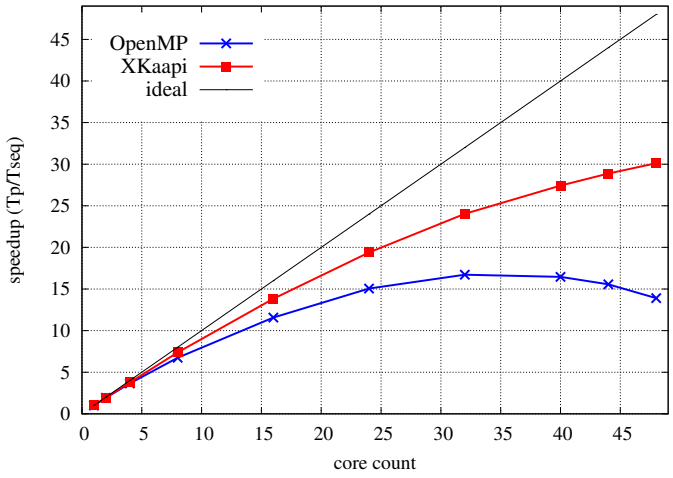
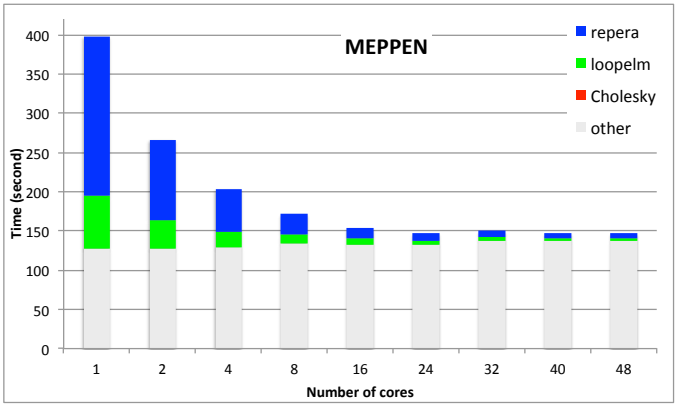


Fig. 7. Sequential sparse Cholesky code (top). Speedups of X-KAAPI vs OpenMP (bottom).

kind of access mode and the memory region shape that is defined : StarSs/SMPSs [3], QUARK have similar access mode and consider unidimensional array. QUARK has an original `scratch` access mode to reuse thread specific temporary data. StarPU [1] has a more complex way to split data and define sub-view of a data structure. X-KAAPI has direct support for multi-dimensional arrays and sub-arrays. In [7] OMPSs data specification will only deals with contiguous memory region. New extension [6] considers non contiguous memory region.

The data flow task model is flat in StarSs/SMPSs, QUARK and StarPU while OMPSs andX-KAAPI allow recursive task creation. The fork-join parallel paradigm is only supported by X-KAAPI, Intel TBB [25], Cilk [13] and Cilk+ (Intel version of Cilk). The X-KAAPI performance for fine grain recursive applications is equivalent, or even better, than Cilk+ and Intel TBB that only allow independent task creations. In TBB, Cilk or X-KAAPI task creation is several order of magnitude less costly than in StartSs/SMPSs, QUARK or StarPU. QUARK and StarPU cannot scale well due to their central list scheduling. SMPSs and OMPSs seem to support a more distributed scheduling.

StarPU [1], OMPSs [7] and X-KAAPI [16] allows to exploit multi-CPUs and multi-GPUs architectures. OMPSs may schedule program on a distributed memory architecture in the same fashion as our old Kaapi implementation [15]. StarPU [2] allows to mix dependent tasks with MPI commu-



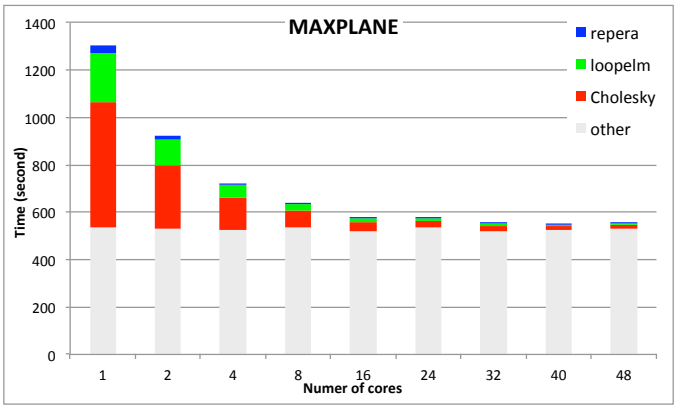


Fig. 8. Overall gains for EPX with X-KAAPI.

nication primitives. X-KAAPI provides a binary compatible libGOMP library called libKOMP [5].

X-KAAPI has a unique model of adaptive task that allow a runtime adaptation of task creation when resources are idle. The OpenMP runtime of GCC 4.6.2, libGOMP, implements a threshold heuristic that limits task creation when the number of tasks is greater than 64 times the number of threads. It can limit the parallelism of the application and thus performance cannot be guaranteed like with a workstealing algorithm. TBB, with autopartitionner heuristic, is able to limit the number of tasks without, a priori, limit the parallelism of the application.

Intel TBB, Cilk+, OpenMP and X-KAAPI support parallel loop which are not present in StarSs/SMPSs, QUARK or StarPU. Our comparison with OpenMP/GCC 4.6.2 shows that for benchmarked instances and applications, scheduling strategy is not an important feature.

VI. CONCLUSIONS AND FUTURE DIRECTIONS

This paper introduced the X-KAAPI multi paradigm parallel programming model. Experiments highlighted that for each paradigm specific benchmark, X-KAAPI reaches a similar or better performance than the reference software for this paradigm. We also compare OpenMP and X-KAAPI on the industrial code EUROPLEXUS. If for the parallel loop parallelism, X-KAAPI and OpenMP show an equivalent performance (with better scalability for X-KAAPI), for data flow tasks the OpenMP parallel model imposes synchronizations that limits the speedup. This overhead experienced with our sparse Cholesky factorization, was already spotted in [19] on dense linear algebra factorizations.

This X-KAAPI evaluation draws two interesting conclusions: 1/ the OpenMP static and dynamic schedulers, which comes from historical design choices, would benefit from being extended to match application characteristics. Intel TBB only proposes a dynamic scheduler. We are investigating the performance of our new adaptive loop scheduler implemented in GCC/OpenMP runtime [11] on the code EUROPLEXUS. 2/ a (macro) data flow task model supporting recursivity can be efficiently implemented and be competitive with a simple fork-join model. It completes our previous published results using multi-CPUs multi-GPUs support [18], [16].

Ongoing work focuses on our compiler infrastructure and distributed memory architecture support as previously experimented [15]. The new OpenMP-4.0 standard (http://www.openmp.org) will allow to program with tasks with (data flow) dependencies. We will investigate this new extension and how to reuse our library in an high performance OpenMP runtime support.

ACKNOWLEDGMENT

This work has been supported by CEA and by the ANR Project 09-COSI-011-05 RePDyn.

REFERENCES

[1] E. Agullo, C. Augonnet, J. Dongarra, H. Ltaief, R. Namyst, J. Roman, S. Thibault, and S. Tomov, "Dynamically scheduled Cholesky factorization on multicore architectures with GPU accelerators." in *Symposium on Application Accelerators in High Performance Computing (SAAHPC)*, Knoxville, USA, 2010.

[2] C. Augonnet, O. Aumage, N. Furmento, R. Namyst, and S. Thibault, "Starpu-mpi: task programming over clusters of machines enhanced with accelerators," in *Proceedings of the 19th European conference on Recent Advances in the Message Passing Interface*, ser. EuroMPI'12. Berlin, Heidelberg: Springer-Verlag, 2012, pp. 298–299. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-33518-1_40

[3] R. M. Badia, J. R. Herrero, J. Labarta, J. M. Pérez, E. S. Quintana-Ortí, and G. Quintana-Ortí, "Parallelizing dense and banded linear algebra libraries using smpss," *Concurr. Comput. : Pract. Exper.*, vol. 21, pp. 2438–2456, 2009.

[4] R. D. Blumofe and C. E. Leiserson, "Space-efficient scheduling of multithreaded computations," *SIAM J. Comput.*, vol. 27, pp. 202–229, 1998.

[5] F. Broquedis, T. Gautier, and V. Danjean, "Libkomp, an efficient openmp runtime system for both fork-join and data flow paradigms," in *Proceedings of the 8th international conference on OpenMP in a Heterogeneous World*, ser. IWOMP'12. Berlin, Heidelberg: Springer-Verlag, 2012, pp. 102–115. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-30961-8_8

[6] J. Bueno, X. Martorell, R. M. Badia, E. Ayguadé, and J. Labarta, "Implementing ompss support for regions of data in architectures with multiple address spaces," in *Proceedings of the 27th international ACM conference on International conference on supercomputing*, ser. ICS '13. New York, NY, USA: ACM, 2013, pp. 359–368.

[7] J. Bueno, J. Planas, A. Duran, R. M. Badia, X. Martorell, E. Ayguadé, and J. Labarta, "Productive programming of gpu clusters with ompss," in *26th IEEE International Parallel and Distributed Processing Symposium, IPDPS 2012*. Shanghai, China: IEEE Computer Society, 2012, pp. 557–568.

[8] A. Buttari, J. Langou, J. Kurzak, and J. Dongarra, "A class of parallel tiled linear algebra algorithms for multicore architectures," *Parallel Comput.*, vol. 35, pp. 38–53, 2009.

[9] B. Dumitrescu, M. Doreille, J.-L. Roch, and D. Trystram, "Two-dimensional block partitionings for the parallel sparse cholesky factorization," *Numerical Algorithms*, vol. 16, pp. 17–38, 1997.

[10] A. Duran, R. Ferrer, E. Ayguadé, R. M. Badia, and J. Labarta, "A proposal to extend the openmp tasking model with dependent tasks," *Int. J. Parallel Program.*, vol. 37, pp. 292–305, June 2009.

[11] M. Durand, F. Broquedis, T. Gautier, and B. Raffin, "An Efficient OpenMP Loop Scheduler for Irregular Applications on Large-Scale NUMA Machines," in *IWOMP*, Canberra, Australia, sep 2013.

[12] F. E. Fich, "New bounds for parallel prefix circuits," in *Proceedings of the fifteenth annual ACM symposium on Theory of computing*, ser. STOC '83. New York, NY, USA: ACM, 1983, pp. 100–109.

[13] M. Frigo, C. E. Leiserson, and K. H. Randall, "The implementation of the cilk-5 multithreaded language," *SIGPLAN Not.*, vol. 33, pp. 212–223, 1998.

[14] F. Galilée, J.-L. Roch, G. G. H. Cavalheiro, and M. Doreille, "Athapascan-1: On-line building data flow graph in a parallel language," in *Proceedings of PACT'98*, ser. PACT '98. Washington, DC, USA: IEEE Computer Society, 1998, pp. 88–.

[15] T. Gautier, X. Besseron, and L. Pigeon, "KAAPI: A thread scheduling runtime system for data flow computations on cluster of multiprocessors," in *Proceedings of PASCO'07*. New York, NY, USA: ACM, 2007.

[16] T. Gautier, J. V. Ferreira Lima, N. Maillard, and B. Raffin, "XKaapi: A Runtime System for Data-Flow Task Programming on Heterogeneous Architectures," in *In Proc. of the 27-th IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, Boston, USA, jun 2013.

[17] D. Hendler, I. Incze, N. Shavit, and M. Tzafrir, "Flat combining and the synchronization-parallelism tradeoff," in *Proceedings of the 22nd ACM SPAA*. New York, NY, USA: ACM, 2010.

[18] E. Hermann, B. Raffin, F. Faure, T. Gautier, and J. Allard, "Multi-GPU and Multi-CPU Parallelization for Interactive Physics Simulations," in *EUROPAR 2010*, Ischia Naples, Italy, 2010.

[19] J. Kurzak, H. Ltaief, J. Dongarra, and R. M. Badia, "Scheduling dense linear algebra operations on multicore processors," *Concurr. Comput. : Pract. Exper.*, vol. 22, pp. 15–44, 2010.

[20] E. A. Lee, "The problem with threads," *Computer*, vol. 39, pp. 33–42, 2006.

[21] F. Lementec, V. Danjean, and T. Gautier, "X-Kaapi C programming interface," INRIA, Tech. Rep. RT-0417, 2011.

[22] F. Lementec, T. Gautier, and V. Danjean, "The X-Kaapi's Application Programming Interface. Part I: Data Flow Programming," INRIA, Tech. Rep. RT-0418, 2011.

[23] OpenMP Architecture Review Board, "http://www.openmp.org," 1997-2008.

[24] D. Quinlan and *et al*, "Rose compiler project." [Online]. Available: http://www.rosecompiler.org

[25] J. Reinders, *Intel threading building blocks*, 1st ed. Sebastopol, CA, USA: O'Reilly & Associates, Inc., 2007.

[26] M. Tchiboukdjian, N. Gast, D. Trystram, J.-L. Roch, and J. Bernard, "A Tighter Analysis of Work Stealing," in *The 21st International Symposium on Algorithms and Computation (ISAAC)*, no. 6507. Jeju Island, Korea: Springer, 2010.

[27] D. Traore, J.-L. Roch, N. Maillard, T. Gautier, and J. Bernard, "Deque-free work-optimal parallel STL algorithms," in *EUROPAR 2008*. Las Palmas, Spain: Springer-Verlag, 2008.

[28] A. YarKhan, J. Kurzak, and J. Dongarra, "Quark users' guide: Queueing and runtime for kernels," University of Tennessee, Tech. Rep. ICL-UT-11-02, 2011.